SATC's Interpretation Guidelines

for C++ Object Oriented Metrics

Object Oriented Metrics - Definitions

The SATC's set of object oriented metrics focuses on internal object structures that reflect the complexity of each individual entity, such as methods and classes, and on external complexity that measures the interactions among entities, such as message passing, coupling, cohesion, and hierarchical structure. A brief description of object oriented terms is given below with the interpretation guidelines used by the SATC for C++.

A C++ class is like a template from which objects can be created. This set of objects shares a common structure and a common behavior manifested by the set of methods. A method is an operation upon an object and is defined in the class declaration. Methods consist of attributes and procedures. Classes are in a hierarchical structure, with children inheriting methods and attributes from parent classes.

Number of Methods per Class is a count of the methods defined within each class. In NASA object oriented code we usually find that classes have 1 - 10 methods, while acknowledging that 20 - 40 methods are occasionally necessary. More than 40 methods per class is an indication of "overloading" the classes, too much activity in one class and can indicate improper or inefficient use of object oriented structures. Classes with large numbers of methods are likely to be more application specific and not reusable. The expected distribution of this data is shown in Figure 1. The percentages on this graph and other guideline graphs are indicators only. Percentages are approximates only. The number on the far right on the x-axis is an indicator of the highest value expected.

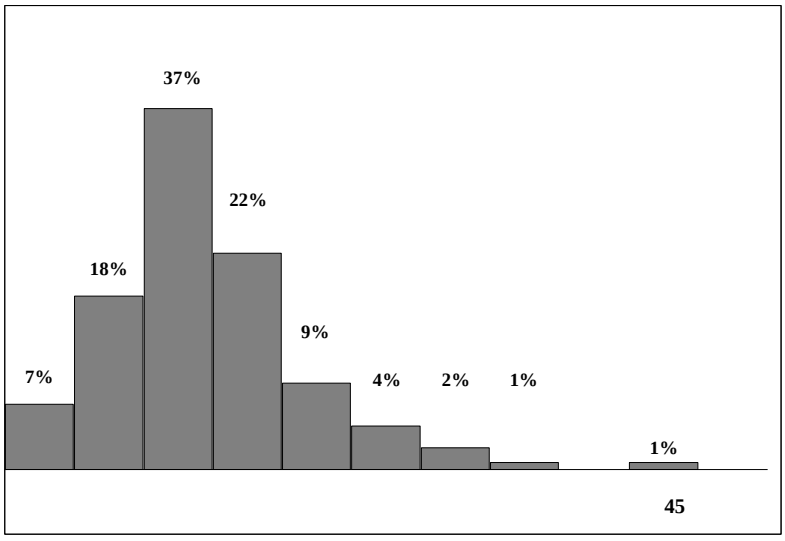

Figure 1: Expected Curve – Number of Methods in a C++ class

Weighted Methods per Class (WMC) is the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). A method with a low cyclomatic complexity will need less testing and will be easier to understand. A low WMC can result when decisions are deferred through message passing, disguising a method that is complex. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children; children inherit all of the methods defined in the parent class. The C++ classes with the highest WMC are candidates for inspection and/or revision.

Because the recommended complexity of a method is 5 or less, and classes should have less than 20 methods, the WMC should be below 100. The maximum number of methods per class is 40 so WMC should be less than 200. The shape of the right hand tails of the histograms for Number of Methods and Weighted Methods per Class should be rapidly diminishing if there are no classes with disproportionate complexity per method.

Response for a Class (RFC) is the count of the set of all unique methods, inside and outside a class, that can be invoked in response to a message to an object of the class or by some method in the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated because it requires a greater level of understanding on the part of the tester. RFC serves as a worst case value for possible responses and is useful information for estimating testing. Figure 2 is an example of the distribution expected for RFC.

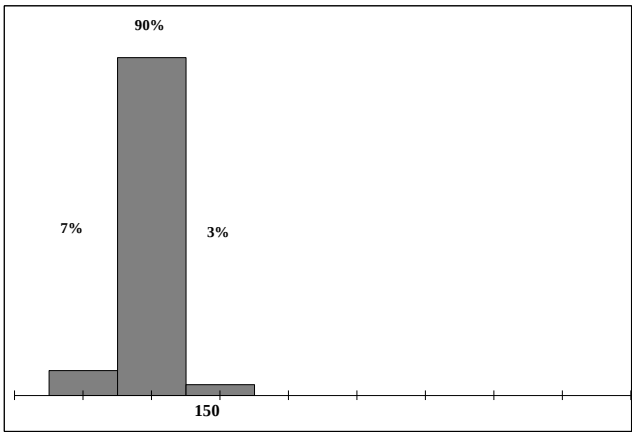

Figure 2: Expected Curve - Response for Class

Another way to look at the RFC is to combine it with the number of methods in Figure 3. This information identifies those classes that are very big and "influence" or "touch" a high number of other classes. If these modules are changed, the potential impact is very high and hence the risk of a "bad fix" or errors resulting from the change is very high. This graph is developed by first plotting a point for each class's RFC versus its Number of Methods. Then calculate the fitted "average" (the dashed line,) then for reference, add a solid line at 5 times the number of methods. Based on the SATC experience, 5 times the number of methods gives a good maximum guideline, above which there are usually just a few classes that make a lot of

calls to other classes. The bottom solid line is where the RFC is equal to the number of methods. A vertical line is drawn at 40 because the number of methods in a class should not exceed 40. Those classes that are in the upper right area are large and interconnected and need a great deal of testing. In general, few classes should be above the top line, and the lower the dashed "average" line, the better.

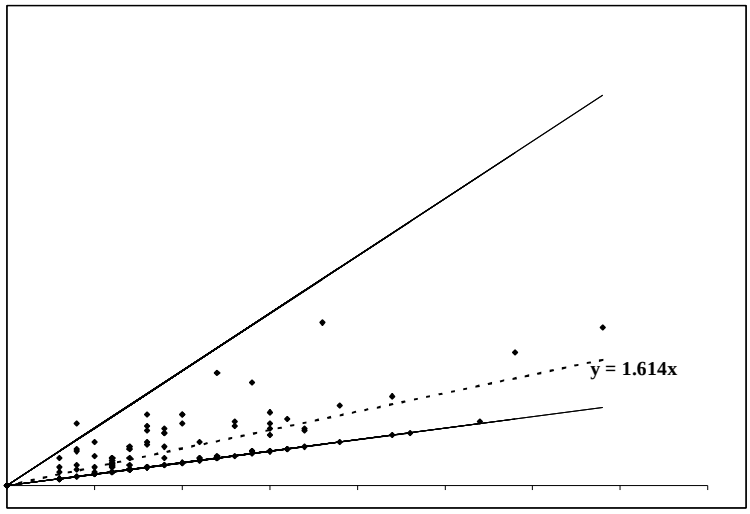

Figure 3: Example - Response for Class vs. Number of Methods

Coupling Between Object Classes (CBO) is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct, non-inheritance related class hierarchies upon which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, therefore making maintenance more difficult. Strong coupling complicates a system because a class is harder to understand, change or correct by itself if it is interrelated with other classes. Designing systems with the weakest possible coupling between classes can reduce complexity. This improves modularity and promotes encapsulation. Higher CBO indicates classes that may be difficult to understand, less likely for reuse and more difficult to maintain. Figure 4 is an indication of the expected histogram of this metric. Generally, coupling between objects should be below 6, may go as high as 10 and still be acceptable, but should not exceed 10. Figure 4 shows the expected shape of the distribution of the CBO metric.

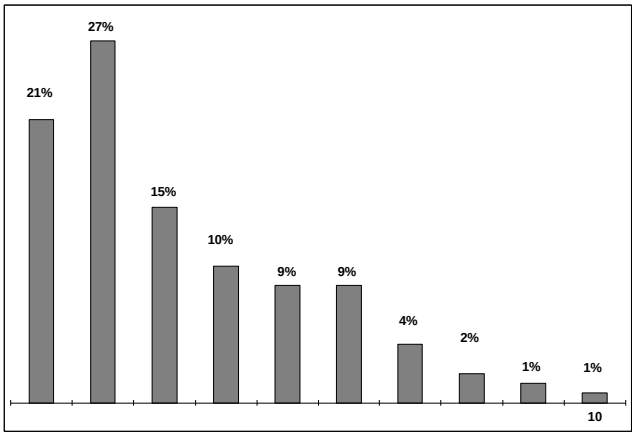

Figure 4: Expected Coupling Between C++ Objects

The **Depth of Inheritance Tree (DIT)** is the depth of a class within the inheritance hierarchy and is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number methods it is likely to inherit making it more complex to predict its behavior. Deeper trees constitute greater design complexity, because more methods and classes are involved, but the greater the potential for reuse of inherited methods.

The **Number of Children (NOC)** is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of subclassing. But the greater the number of children, the greater the reuse because inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time.

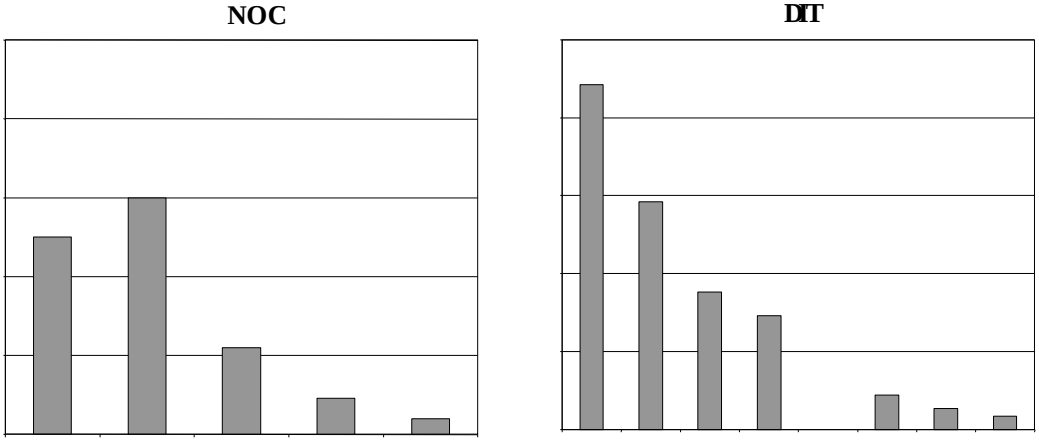

Figure 5: NOC and DIT for a Typical Object Oriented C++ System